PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 11/30, 9/00 — A1

(11) International Publication Number: WO 97/24671

(43) International Publication Date: 10 July 1997 (10.07.97)

(21) International Application Number: PCT/US96/20126

(22) International Filing Date: 23 December 1996 (23.12.96)

(30) Priority Data:
08/578,203 29 December 1995 (29.12.95) US

(71) Applicant: POWERTV, INC. [US/US]; Suite 100, 20833 Stevens Creek Boulevard, Cupertino, CA 95014 (US).

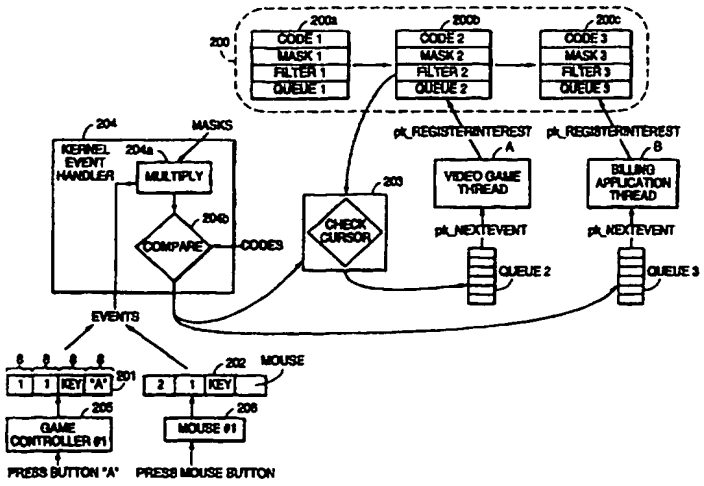
(72) Inventor: HOUHA, James, A.; Powertv, Inc., Suite 100, 20833 Stevens Creek Boulevard, Cupertino, CA 95014 (US).

(74) Agents: POTENZA, Joseph, M. et al.; Banner & Witcoff, Ltd., 11th floor, 1001 G Street, N.W., Washington, DC 20001-4597 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: EVENT FILTERING FEATURE FOR A COMPUTER OPERATING SYSTEM IN A HOME COMMUNICATIONS TERMINAL



(57) Abstract

An improved operating system kernel for a home communication terminal (HCT) includes an event filtering feature (204) which allows threads (A, B) running in the HCT to register interest in events of a particular type, from a particular source, or other desirable criteria. Events occurring in the system (205, 206) are prequalified by the kernel before providing them to individual threads which have registered interest in only certain types of events. By executing a filter in the kernel's context, a thread context switch can be avoided. Events occurring in the system can be matched with events of interest (200a, 200b, 200c) registered by various threads by an efficient comparison operation including a mask field and a code field. Additionally, various thread synchronization mechanisms such as alarms and semaphores can be implemented using a common event object which is integrated onto event queues.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	**GB**	United Kingdom	**MW**	Malawi
AT	Austria	**GE**	Georgia	**MX**	Mexico
AU	Australia	**GN**	Guinea	**NE**	Niger
BB	Barbados	**GR**	Greece	**NL**	Netherlands
BE	Belgium	**HU**	Hungary	**NO**	Norway
BF	Burkina Faso	**IE**	Ireland	**NZ**	New Zealand
BG	Bulgaria	**IT**	Italy	**PL**	Poland
BJ	Benin	**JP**	Japan	**PT**	Portugal
BR	Brazil	**KE**	Kenya	**RO**	Romania
BY	Belarus	**KG**	Kyrgystan	**RU**	Russian Federation
CA	Canada	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SG**	Singapore
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovakia
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LR**	Liberia	**SZ**	Swaziland
CS	Czechoslovakia	**LT**	Lithuania	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**LV**	Latvia	**TJ**	Tajikistan
DK	Denmark	**MC**	Monaco	**TT**	Trinidad and Tobago
EE	Estonia	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**UG**	Uganda
FI	Finland	**ML**	Mali	**US**	United States of America
FR	France	**MN**	Mongolia	**UZ**	Uzbekistan
GA	Gabon	**MR**	Mauritania	**VN**	Viet Nam

EVENT FILTERING FEATURE FOR A COMPUTER OPERATING SYSTEM IN A HOME COMMUNICATIONS TERMINAL

BACKGROUND OF THE INVENTION

1. Technical Field

This invention relates generally to real-time operating systems adapted for high performance applications, such as those executing in a home communications terminal (HCT) to provide cable television or other audiovisual capabilities. More particularly, the invention provides a feature which improves the performance of operating systems installed in devices having limited computing resources.

2. Related Information

Conventional operating systems for HCTs, such as those in cable television systems, have typically provided limited capabilities tailored to controlling hardware devices and allowing a user to step through limited menus and displays. As growth in the cable television industry has fostered new capabilities including interactive video games, video-on-demand, downloadable applications, higher performance graphics, multimedia applications and the like, there has evolved a need to provide operating systems for HCTs which can support these new capabilities. Additionally, newer generations of fiber-based networks have vastly increased the data bandwidths which can be transferred to and from individual homes, allowing entirely new uses to be developed for the HCTs. Changing federal and state regulations also portend new uses such as

telephony to be available through existing cable networks. As a result, conventional HCTs and their operating systems are quickly becoming obsolete. In short, HCTs need to evolve to transform today's limited capability television sets into interactive multimedia entertainment and communication systems.

One possible approach for increasing the capabilities of HCTs is to port existing operating systems, such as UNIX or the like, to PC-compatible microprocessors in the HCT. However, the huge memory requirements needed to support such existing operating systems render such an approach prohibitively expensive. Because memory is a primary cost component of HCTs, competitive price pressures mean that the added functions must be provided in a manner which minimizes memory use and maximizes processor performance. Consequently, it has been determined that new operating system features must be developed which provide media-centric high performance features while minimizing memory requirements.

One conventional operating system design paradigm which has been determined to generally consume a large amount of memory is the partitioning of thread coordination mechanisms -- such as semaphores, timers, exceptions, messages, and so forth -- into separate subsystems in the operating system. Each such subsystem conventionally includes different application programming interface (API) conventions, different data structures, and different memory areas used by the kernel to keep track of and to check on them.

Another conventional operating system design paradigm which has been determined to cause operating system inefficiency in a real-time operating system is the manner in which events are transferred to threads executing in the system. Conventional approaches for delivering an event to a thread involve scheduling the thread and providing the event to the thread, even though the event may not be of interest to the thread (i.e., after receiving the event, the thread immediately determines that it is not of interest and discards it). Such a scheme wastes processing time performing a context switch, and also wastes memory space.

As one example, if a user presses a key on an HCT keypad, a conventional kernel would transfer that event to a thread which handles the event, even though the event may be of no interest unless a cursor on the television screen is within a certain window. This results in inefficiency, since executing the thread involves a context switch, followed by the thread quickly determining that the event is of no interest because of the cursor location on the screen. Many other examples of such inefficiency stem from the fact that threads in a system cannot "prequalify" events which they are to receive from the kernel.

In summary, conventional operating systems for use in HCT applications suffer from performance and memory disadvantages which hinder their utility when used for newer, high performance graphics-intensive applications. Accordingly, there is a need to provide operating system features which can reduce memory requirements and simultaneously increase the overall performance of applications executing in conjunction with the operating system.

SUMMARY OF THE INVENTION

The present invention solves the aforementioned problems by providing an efficient real-time kernel having features tailored to the needs of HCT applications. Whereas conventional kernels typically provide separate event subsystems, semaphore subsystems and queue subsystems, one aspect of the present invention contemplates replacing such subsystems with a single, integrated event subsystem which provides the functionality of semaphores and other synchronization mechanisms through events on event queues. Instead of using different data structures to provide these different services, a single event data structure can be used. This single data structure can be optimized to speed up the kernel. Because the kernel needs to be aware of only of two states for each thread (executing the thread, or waiting for an event to deliver to the thread), kernel efficiency can be increased. In contrast, conventional kernels typically require that the kernel distinguish between various other states, such as waiting for a semaphore, an event, message stream, or an I/O operation, thus resulting in increased complexity and increasing memory requirements in the kernel.

Another aspect of the present invention contemplates providing means for each thread to "register" with the kernel to indicate what kind of events (or classes of events) the thread would like to receive. Each thread can also specify a "filter" procedure which, when an event is posted to the system (but before it is delivered to the thread), decides whether the posted event is appropriate for

that context or not. This filter may be an interrupt service routine which runs at interrupt time instead of invoking the destination thread, which would require a context switch.

As used herein, the term "home communication terminal" (HCT) will be understood to refer to terminals that can be used in telephone networks, cable TV or other audiovisual programming networks, satellite networks, or combinations of these.

Various other objects and advantages of the present invention will become apparent through the following detailed description, figures, and the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows one possible configuration for a home communication terminal (HCT) on which an operating system employing the principles of the present invention can be installed.

FIG. 2 shows schematically how a kernel event handler 204 constructed in accordance with the present invention can efficiently handle and filter incoming events.

FIG. 3 shows steps which may be executed by a kernel event handler to efficiently handle events in an HCT.

FIG. 4 shows an example of different threads having registered interest in different types of events.

FIG. 5 shows one possible format for an event object in a system employing the principles of the present invention.

FIG. 6 shows how two threads can implement a semaphore by using a queue wherein a kernel implements a NextEvent function.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIG. 1 shows a block diagram of a home communication terminal (HCT) in which various principles of the present invention may be practiced. The HCT may include a CPU card 100, graphics card 101, decoder card 102, display panel and key pad 103, main processing board 104, front end 105, tuning section 106, and audio section 107. It is contemplated that the inventive principles may be practiced using any suitable CPU 100a such as a PowerPC or Motorola 68000 series, with suitable EPROM 100c and RAM 100b. It is also contemplated that application programs executing on CPU 100a can interact with various peripherals such as a mouse, game controllers, keypads, network interfaces, and the like, as is well known in the art.

FIG. 2 shows schematically how a kernel event handler employing various inventive principles can efficiently handle incoming events and prequalify the events to certain threads executing in the HCT. Generally speaking, different threads in the system may be interested only in certain types of events, and may wish to ignore other types of events. Examples of events include: a keypress on a keypad attached to the HCT; a mouse movement indication received from a mouse attached to the HCT; a button press on a game controller connected to the

HCT; a message received from a headend coupled to the HCT; or a signal indicating that a movie has started. Examples of different threads (which may be concurrently executing in a multi-threaded kernel) include a copy of a video game operated by a first player; a copy of a video game operated by a second player; an on-screen programming guide; a movie player; a channel tuning indicator; or a user interface for a customer billing application. In general, one application may correspond to a single thread, or a single application may be partitioned into multiple threads which may be concurrently executed to optimize performance. One of ordinary skill in the art will recognize how various applications may be constructed out of a single or multiple threads in the system.

A customer billing application may only be interested in events occurring from the HCT keypad, and only if the customer has first entered a special code. Two video game threads may be interested in all key press events occurring from either of two game controllers coupled to the HCT (thus requiring copies of the same event to be posted to both threads). An on-screen programming guide thread may only be interested in keypress events which occur when the mouse cursor is positioned within a certain predetermined area on the screen, and to ignore all other events (including keypress events when the mouse cursor is not positioned within the area). Many other examples are of course possible. Each thread may thus wish to register interest in a plurality of different types of events, and may wish to change the registered interests at a later time.

FIG. 2 shows an exemplary configuration including a kernel event handler 204 which can accomplish the above objectives. As shown in FIG. 2, two threads A and B can each register interest with the kernel in two different events using a function pk_RegisterInterest (see Appendix 1), a kernel-provided programming interface. A corresponding function pk_RemoveInterest allows a thread to remove an interest in a thread (see Appendix 1). In response to invocations of pk_RegisterInterest, the kernel constructs an event interest list 200 which may comprise a linked list of "event interest" objects 200a, 200b, and 200c. For example, thread A may register interest in an event corresponding to event interest 200b, and thread B may register interest in an event corresponding to event interest 200c, each of which are specified by parameters in corresponding function calls to pk_RegisterInterest. It will be assumed for the example in FIG. 2 that thread A comprises a video game application which is only interested in key press events from game controllers, while thread B comprises a billing application which is only interested in button presses from a mouse.

In various embodiments, the kernel manipulates event interest list 200 in response to calls to pk_RegisterInterest and pk_RemoveInterest. When kernel event handler 204 receives or generates an event, it traverses event interest list 200 to determine the conditions under which various threads in the system should be invoked. In general, by comparing an event descriptor (which describes the event) with parameters included in each event interest object, kernel event

handler 204 can efficiently determine whether and how to invoke any thread which has expressed interest in an event.

Each event interest object 200a, 200b, and 200c may comprise a code field, a mask field, a filter procedure field, and a queue field in accordance with the parameters set forth for pk_RegisterInterest. For example, when thread A registers interest in certain game controller events, it specifies code2, mask2, filter2, and queue2 as parameters in function pk_RegisterInterest.

Reference will be made briefly to Appendix 1, which includes descriptions for a plurality of functions which may be used to carry out various principles of the invention. For each function in Appendix 1, a syntax including a list of parameter types and examples of use is provided. Referring to function pk_RegisterInterest, for example, thread A would invoke this function and supply parameter values for code, mask, filter, and queue (the filter and queue parameters are optional) in order to direct the kernel to prequalify and direct events to thread A.

The pk_RegisterInterest function (see Appendix 1) creates and registers an event interest with the kernel. Its code parameter specifies a description of the desired event, its mask parameter specifies an event mask which further clarifies events of interest, its filter parameter specifies an interrupt service routine (ISR) for the kernel to call when the event occurs, and its queue parameter specifies a pointer to the queue to which to route events.

Each event interest object, such as element 200b in FIG. 2, holds a mask and a code specifying a "desirable" event descriptor for an incoming event. The mask specifies which fields of the event descriptor are germane to the specified interest, and the code specifies the values that those fields must have in order for the posted event to trigger the event interest.

Bits in the mask field can be used to indicate which fields of the event descriptor are germane to a particular interest. One possible example is to allocate 32 bits to the mask field. Of the 32 bits, 8 bits can be allocated to indicate the device type (i.e., the device type which generated the event), another 8 bits for the device instance (i.e., the instance of that device type which generated the event), another 8 bits for the event type (i.e., what type of event the device generated, such as a key press), and another 8 bits for event data (i.e., a small amount of data which can contain the event information, such as the specific key which was pressed). Such an allocation is by way of example only, and is not intended to be limiting.

Corresponding fields may be included in each event descriptor, as indicated by event descriptor 201 in FIG. 2, for example. Thus, each event descriptor can include an indication of the device type, device instance, event type, and event data corresponding to the event.

To register interest in events from any device type, a thread would set the device type parameter to all ones. The following (hex) masks could thus be defined:

kDt_Any	00FFFFFF	any device type (bits 24 to 31)
kDi_Any	FF00FFFF	any device instance (bits 16 to 23)
kEt_Any	FFFF00FF	any event type (bits 8 to 15)
kEd_Any	FFFFFF00	any data values (bits 0 to 7)

For example, to register an interest in all game controller events, an application would call:

pk_RegisterInterest (kDt_Controller,

kDi_Any & kEt_Any & kEd_Any, NULL, my_Q)

where kDt_Controller is a code corresponding to the game controller, and the event mask multiplies to the value FF000000 (no filter procedure was specified).

As can be seen, the mask FF000000 would cause all the device type bits to be set, forcing a comparison of device type with that specified in the code (i.e., Controller type), but leaving zero ("not caring about") the device instance (the next 8 bits), event type (the following 8 bits), or the event data (the last 8 bits). It will be appreciated that an event descriptor could be created with subsets of the above fields, or with entirely different fields which qualify a particular type of event.

Returning to the example in FIG. 2, suppose that thread A wishes to register interest in all game controller key press events which occur when the cursor is within a predetermined screen area, and to ignore all other types of events. Thread A creates an event interest 200b by specifying mask2 which specifies "device type" as a qualifier, leaves the device instance open, specifies

"event type" as a qualifier, and leaves the "event data" open. Additionally, thread A specifies code2 which identifies "game controller" as the device type and "key press" as the desired event type. Finally, thread A specifies a filter procedure 203 which is to be executed to further qualify the event, and a queue onto which the event will be placed (queue2). For the example shown in FIG. 2, filter procedure 203 checks the cursor location to ensure that it is within a qualified area before passing on the event. It is assumed that filter procedure 203 executes in kernel mode, thus avoiding a context switch. In other words, thread A will not be scheduled by the kernel unless the event meets all of thread A's qualifications.

When an event from game controller 205 is generated, an event descriptor 201 is created which corresponds to the particulars of the event. A device driver, which detects a hardware change, can post such an event using pk_PostEvent (see Appendix 1). For the example in FIG. 2, the "A" button on game controller 205 has been pressed, and event descriptor 201 thus indicates the device type as game controller ("1"), the device instance as "1", the event type as "key", and the event data as "A", corresponding to the "A" button. Kernel event handler 204 receives the incoming event (from pk_PostEvent), and traverses event interest list 200 to match the incoming event to events of interest.

In a preferred embodiment, the event matching step may be performed very efficiently by multiplying the mask of each event interest object with the incoming event descriptor, then comparing the result with the code of the event

interest object. If there is a match, then the event has been pre-qualified. In FIG. 2, for example, kernel event handler 204 first examines event interest 200a by multiplying event descriptor 201 with mask1 and comparing the result to code1, and quickly determines that there is no match. This "AND" then "COMPARE" operation can be done very efficiently on a CPU (typically, only two assembly language instructions are needed), thus adding to the performance increase which results from using the principles of the invention.

After determining that event descriptor 201 does not match event interest object 200a, kernel event handler 204 next examines event interest object 200b and multiples mask2 by event descriptor 201, then compares the result with code2. For the example in FIG. 2, assume that the result is a successful match. However, because thread A specified a filter procedure 203, kernel event handler 204 does not pass the event to thread A but instead executes filter procedure 203, which checks the cursor location to see if it is in a valid area. If it is, kernel event handler 204 delivers the event to queue2 (also specified in event interest object 200b), and thread A can extract the event from queue2 using pk_NextEvent (see Appendix 1).

It should be noted that had the event not successfully passed through filter procedure 203, thread A would not have been scheduled, thus avoiding a context switch and increasing the efficiency of the kernel. In other words, in a preferred embodiment, filter routine 203 is executed while the kernel is executing, and need not schedule thread A. If only 20% of events are actually of interest to a

particular thread, then it is much faster to make the "interest" determination in kernel code than in thread code, which requires context switches.

Continuing with the example in FIG. 2, suppose that thread B has registered interest in any mouse movement from mouse 206. It would have set up event interest object 200c by way of pk_RegisterInterest, specifying device type as a qualifier, but leaving the other mask fields open. For device type in code3, thread B would have specified "mouse". When a user presses a mouse button, event descriptor 202 would be generated, indicating device type as mouse ("2"), device instance "1", event type as "key", and event data as "mouse press". Kernel event handler 204 would traverse event interest list 200, preferably multiplying each mask with the event descriptor and comparing the result with the code. When it reached event interest object 200c, it would find a match, and immediately place the event on queue3, which was specified by thread B as the desired queue (no filter procedure was specified).

FIG. 3 shows steps which may be executed by kernel event handler 204 to handle incoming events in the system. It is assumed that an event interest list has already been created through the use of pk_RegisterInterest calls made by threads in the system. Beginning in step 301, the next event interest object from the event interest list is retrieved. In step 302, the mask from the event interest object is multiplied with the event descriptor corresponding to the event. In step 303, the result is compared with the code from the event interest object. In step 304, if there is no match, processing resumes at step 301 with the next event

interest object.

If, in step 304, there is a match, then in step 305 a check is made to determine whether a filter was specified for the event interest object. If so, then in step 306 the specified filter is called, preferably directly by the kernel and without a context switch. If the event passes the filter in step 307, processing advances to step 308; otherwise, processing resumes at step 301 with the next event interest object. In step 308, if a queue was specified with the event interest, the event is placed on the specified queue in step 309; otherwise, processing resumes at step 301 until the end of the event interest list is reached.

Note that if no queue was specified, the event can be discarded. Such a situation may be desirable, for example, where all of the processing is done in the filter itself. For example, if the only thing to be done is to update a memory area or other type of short-lived operation, then it may be more efficient to do this in the filter itself (in the kernel), and no queue need be provided (i.e., no thread to wake up).

Additionally, filters can be reusable. For example, one filter might have a function of determining whether a cursor is in a particular location on the screen; different threads could specify and use this same filter. If, for example, three different applications are executing in the HCT, each having a separate window on a television screen, and the user moves a mouse over the screen, a single filter can be devised which determines whether the cursor is over a window boundary for the specified thread.

Furthermore, when both a filter and a queue are specified, the filter can modify the event itself. For example, a filter could change the time of the event before putting it on a queue. This could be used, for example, by a "replay" filter which changes the time stamp on an event to the current time (e.g., translate to current time). Another example involves checking a signature on an event before waking up a thread.

FIG. 4 shows another example which applies the principles of the present invention. Suppose a video game to be executed on an HCT provides two characters who fight one other, with two users each interacting with a separate game controller 401 and 402. The video game may comprise a main video game thread 403 which controls overall scoring and game operation, and two player threads 404 and 405. In the example of FIG. 4, for example, player 1 thread 404 controls the actions of a first character on the video screen, while player 2 thread 405 controls the actions of a second character on the video screen. Thus, thread 404 must manipulate the first video character based on actions taken by player 1, and thread 405 must manipulate the second video character based on actions taken by player 2. Consequently, player 1 thread 404 registers interest in all events from game controller #1 (by specifying the appropriate code, mask, filter and queue parameters), and player 2 thread 405 registers interest in all events from game controller #2.

In the design shown in FIG. 4, suppose that player 1 thread 404 wants to know when player 2 has pressed a "pause" key on game controller #2, and that player 2 thread 405 wants to know when player 1 has pressed a "pause" key on game controller #1 (in other words, either player, including the one operating the opposite controller, can pause the game). In order to accomplish this, player 1 thread 404 can selectively register interest only in "pause" events from game controller #2, and player 2 thread 405 can selectively register interest only in "pause" events from game controller #1. This avoids having the operating system send all events from the opposing game controller to the thread, thus greatly increasing efficiency. If each queue (and each corresponding thread) in FIG. 4 were provided with a copy of every event generated by each game controller, much time would be wasted in processing undesirable events, wasting CPU time and memory. By allowing each thread to separately register interest only in certain types of events and causing the kernel to only send those types of events to each thread, significant performance increases can be achieved.

Similarly, suppose that main video game thread 403 is interested in receiving "game events" from each player thread, and registers accordingly. Each player thread receives events from main video game thread 403, such as a command to cause the character to die because of too many blows by the other player. The aforementioned registered interests are indicated by arrows in FIG. 4, annotated with the type of events registered.

Additionally, suppose that instant replay thread 406 provides a capability to review all previous actions over a time window. Therefore, it registers interest in all events generated by either game controller, and thus gets its own copy of each event generated by the game controllers. This avoids individual threads from having to send "copies" of events between themselves.

As yet another example, an on-line TV guide may cause tuning tables to be downloaded at unknown times. When a new tuning table is downloaded, this could constitute an event. Typically, more than one thread in the HCT may need a copy of this tuning table. Thus, there becomes a problem of determining when to free up memory used to store the tuning table. This can be accomplished by creating a filter which creates a private copy of the tuning table for each thread that registers an interest in the tuning table event. Thus, when all threads are done using their copies, they will destroy their copy, avoiding the problem of determining when the memory area(s) can be freed.

Note that a single event can trigger more than one filter, and can also trigger more than one thread. For example, one thread can be an event recorder (debugger); it would want to obtain a copy of every event in the system. To accomplish this, the thread would create a mask which clears all the bits, indicating interest in any event.

The following describes in more detail how thread synchronization functions, such as semaphores, timers, media events, exceptions, messages, and so forth, can be replaced with event objects and integrated onto an event queue.

to minimize memory requirements and applications development in accordance with another aspect of the invention.

It is contemplated that the kernel deals with time in a very accurate sense. A clock generates time ticks at a very high rate of speed (e.g., 25 MHz), and this clock can be used for timing. In accordance with various aspects of the invention, every event in the system can include a time stamp (comprising, e.g., 64 bits), comprising a "snapshot" of the system clock. When a event is posted or delivered, a future time can be inserted into the event object (instead of the current time). Rather than immediately delivering the event to a destination thread, the kernel can hold onto the event and not post it until the designated time arrives. Therefore, every event has the capability of being an alarm. For example, every keypress generates an event at a particular time.

Instead of writing an alarm procedure which detects that a certain time has arrived (to schedule an event), a future time can be inserted into an event object, such that when the event is posted, it will not actually be delivered until the future time in the event object. This avoids the need for kernel code to implement alarms, including the attendant data structures, storage areas, and status checks indicating which of several states a thread is in (e.g., "waiting for alarm").

FIG. 5 shows one possible configuration for an event object 501 in a kernel, including a pointer to the next event object, a code (corresponding to an event descriptor), time, X, Y, Z fields, and a "where" pointer. In various

embodiments, event object 501 may comprise 28 bytes including both "public" portions accessible by threads and "private" portions hidden from threads.

Events may be strung together into a list in an "intrusive" form (i.e., the objects in the list have in their data structure the fields strung together), as compared to an "extrusive" form in which the list component is built separately. The "where" field may comprise a queue pointer which is used for scheduled_delivery: deliver event to queue at a future time. A thread may set the code field (device type, instance, etc.) using pk_DeliverEvent where an event is created by a thread; alternatively, a device driver may set the code field using pk_PostEvent as described previously. The X, Y, Z fields comprise "payload"; any data (including pointers to data structures) can be included therein.

As one example, an event structure can be used to set up an alarm. A game might have a 3 minute round. At end of the round, a bell will ring. A thread can set an alarm by calling pk_ScheduledDelivery (see Appendix 1) and set the time to 3 minutes from now, and specify the event code, X, Y, Z, and the thread's own queue as the "where" pointer. The thread can then use pk_NextEvent to get the next event off its queue. Even though different API calls are used, a single small data structure can be used, removing the need for a separate alarm system and eliminating the need for the kernel to provide special functions for alarms. Note that a thread can "pre-timestamp" the object, then the kernel can reuse that same space when the actual time is stamped.

Consolidating alarm functions into an event delivery function thus saves memory, processing time, and programmer effort. When a thread is "sleeping", the kernel need not determine whether the thread is waiting for a semaphore, or for an alarm, or any other type of synchronization event. This results in faster thread switches. Whenever a thread is sleeping, the operating system is always waiting for an event on its queue. This also makes the kernel smaller. In contrast, conventional kernels need to execute instructions to determine whether a thread is waiting for an alarm, semaphore, queue wait, message wait, etc.

A second example of consolidation involves semaphores (FIG. 6). Assume that two threads 602 and 603 need to use a single printer. A conventional approach is to provide a semaphore to which both threads are responsive (i.e., they are in a "semaphore wait" state, and the kernel puts a thread on a "semaphore wait" queue with a semaphore wait data structure).

In contrast to conventional methods, the present invention contemplates using an event object. To implement a semaphore using event queues, a queue 604 is created (typically by printer driver 601) and a single event 604a is placed on the queue. The first thread 602 which needs the resource (such as a printer) makes a call to pk_NextEvent (see Appendix 1), specifying the printer semaphore queue 604. This function causes a wait in the thread if there is no event on the queue, but otherwise extracts the event if there is one on the queue. Thus, assuming the object is still on the queue, the kernel 605 removes it from queue 604 and delivers it to thread 602. If another thread 603 attempts to use the

resource (by calling pk_NextEvent), it will cause the thread to wait until the event 604a is returned to queue 604 by first thread 602. When first thread 602 is finished using the resource, it calls pk_ReturnEvent (see Appendix 1) which returns the event to queue 604, allowing second thread 603 to finally execute its pk_NextEvent function.

Thus, in a preferred embodiment, kernel 605 does not place second thread 603 in a "waiting for semaphore" or "waiting for alarm" or any other type of mode; instead, the common event paradigm is used. This increases the efficiency of the kernel because the kernel need not maintain separate queues for all types of different activities, and when examining a thread's status, the kernel already knows that the thread can be in only one state. Furthermore, the same common event object can be used to implement semaphores in the system; no special semaphore data object needs to be defined.

The above example can be extended to handle multiple resources. For example, in FIG. 6, if there is a second printer 606 (but potentially more than two threads which would need to use the two printers), then each printer could post an event onto queue 604; the first two threads which called pk_NextEvent would obtain use of the printers, and the third caller would be put in a waiting-for-event state by the kernel. This has a further advantage in that no pre-set limit on the number of semaphores needs to be specified; the queue can grow as long as needed. An event can thus be used for anything; the kernel need not even be cognizant of a "semaphore" synchronization mechanism.

To eliminate redundancy and simplify the event system, all of the following types of coordination mechanisms can be classified as events and implemented using functions such as those shown in Appendix 1:

-- thread synchronization, including semaphores, messages, and queue messages, all of which are passed between threads as synchronization objects.

-- demand scheduling, which signals that a demand was made for the upgrade of a thread's priority.

-- timer events and delayed actions (an event can be posted immediately or at a specified future time)

-- inter-thread exception handling (an event occurs whenever a thread raises an exception)

-- user interface actions, including pressing a button on a remote or game controller, changing the volume, moving a pointer device, etc.

-- media events, including starting the playback of audio, reaching the end of a move, inserting media into or ejecting media from a device, etc.

-- application-specific events (user-defined events)

It is apparent that many modifications and variations of the present invention are possible, and references to specific values are by example only. For example, although references herein are made to "threads", such a term should be understood to also include processes, tasks, or other entities which can be scheduled by an operating system. As another example, although application programming interfaces are described for performing various functions such as registering interest in an event, equivalent mechanisms are of course possible to implement the same function. Moreover, specific function names are by way of example only. It is, therefore, to be understood that within the scope of the appended claims the invention may be practiced otherwise than as specifically described.

pk_DeleteQueue

Deletes an event queue.

Syntax

```
void pk_DeleteQueue(queue *q);
```

Parameters

→*q* A pointer to the queue to delete.

Returns

None

Comments

This function not only deletes the specified queue, but frees all the events in the queue as well.

Exceptions

None

Example

None

See Also

pk_FlushQueue, pk_NewQueue

pk_DeliverEvent

Places an event directly into a queue without processing the event interest list.

Syntax

```
void pk_DeliverEvent(queue *q, ui32 code, i32 x, i32 y,
   i32 z);
```

Parameters

→ *q*	A pointer to the queue to which to deliver the event.
→ *code*	The event type code.
→ *x*	An optional event data field.
→ *y*	An optional event data field.
→ *z*	An optional event data field.

Returns

None

Comments

None

Exceptions

kPtv_MemoryFullErr

Example

```
void EventPoster (void *data)
{
   ui32   dtCustomEvent = 0x29000000;

   do
   {
      // Every 5 seconds post an event
      // Alternate between 'posting' and 'delivering'

      pk_SleepFor (1000 * 25000);  // Sleep for one second

      // Broadcast an event, including some arbitrary data
```

```
            pk_PostEvent (dtCustomEvent, 0x8, 0x29, 0x65);

            pk_SleepFor (1000 * 25000); // Sleep for one second

            // Post an event to a specified queue
            pk_DeliverEvent (myQueue, dtCustomEvent, 0x12, 0x14, 0x92);

        } while (1);
    }
```

See Also

pk_ScheduleEvent

pk_EndCatch

Marks the end of a try-and-catch block series.

Syntax

```
pk_EndCatch
```

Comments

This is a macro.

If an exception has not been caught when program control reaches this macro, the exception is thrown to an outer try block. The outer try block can belong to an application, a device, or the operating system itself.

Example

This is a very simple example of an exception handler. The try block surrounds the PlayHighChimeSound routine, and a single catch block catches all exceptions.

```
void PlayHighChimeSound (void)
{
    ui32 playerPtr;

    pk_Try
    {
        // Create an audio player that deletes the resources upon
        // completion of playing the sound.
        playerPtr = ap_NewRAMAIFF ((ui8 *)High_Chime,
                    High_Chime_size, TRUE);
        // Start playing the sound
        ap_play (playerPtr);
    }
    // Catch all audio exceptions here.
    pk_Catch (idPk_ExceptionAll)
    {
        printf ("Audio error caught in PlayHighChimeSound\n");
    }
    pk_EndCatch
}
```

See Also

pk_AlsoCatch, pk_Catch, pk_CatchCleanup

pk_FlushQueue

Flushes an event queue of all events.

Syntax

```
void pk_FlushQueue(queue *q);
```

Parameters

→ *q* A pointer to the queue to flush.

Returns

None

Comments

This function frees all the events in the specified queue.

Exceptions

None

Example

None

See Also

pk_DeleteQueue, pk_NewQueue

pk_FreeEvent

Frees an event.

Syntax

```
void pk_FreeEvent(event *e);
```

Parameters

→*e*	A pointer to the event to free.

Returns

None

Comments

Because multiple threads might be interested in any one event, the kernel creates a new instance of every event that it routes. To prevent memory leaks, we recommend explicitly disposing of processed events using pk_FreeEvent.

Exceptions

None

Example

This example waits for an event to be received, and then frees the event when the application is done with it.

```
const     TimeValue     kForAllEternity = (0xFFFFFFFF, 0xFFFFFFFF);
event     *anEvent;         // Where gamepad and system events
                            // will be posted
ul32      eventDevice;,     // The device that posted the event
          eventData;        // Data posted in the event
queue     *gAppQueue;       // Input event queue

anEvent = pk_NextEvent (gAppQueue, kForAllEternity);
eventData = anEvent->code & kEd_Mask;
eventDevice = anEvent->code & (ul32) kDt_Mask;
pk_FreeEvent (anEvent);
```

See Also

None

pk_Galloc

Allocates memory from the general heap.

Syntax

```
void * pk_Galloc(size_t size);
```

Parameters

→ *size* The size, in bytes, of the block to allocate.

Returns

A pointer to the allocated block, or NULL if there is insufficient memory available.

Comments

None

Exceptions

None

Example

None

See Also

pk_Gfree

pk_Gfree

Frees a memory block in the general heap.

Syntax

```
void pk_Gfree(void *p);
```

Parameters

→ *p*	A pointer to the memory block to free.

Returns

None

Comments

None

Exceptions

None

Example

None

See Also

pk_Galloc

pk_Launch

Creates a new thread.

Syntax

```
Boolean pk_Launch(void (*code)(void *d),
   size_t stk_size, void *data, ui32 priority);
```

Parameters

→ *code*	A pointer to the code to be executed by the new thread.
→ *d*	A pointer to a parameter for *code*.
→ *stk_size*	The size, in bytes, of the stack to allocate for this thread.
→ *data*	A pointer to a private data area.
→ *priority*	The thread priority. The lower 5 bits represents the priority, which can range from 0 to 31.

Returns

TRUE if the thread was launched successfully, and FALSE if the specified priority is already in use by another thread.

Comments

No function exists for deleting the thread because the thread is automatically deleted when the function specified by *code* returns.

Exceptions

kPtv_MemoryFullErr
kPtv_ParamErr

Example

This example launches a new thread on the next available thread number (priority).

```
void LaunchOnNext (void)
{
    ui8     priority = 16;

    while (pk_Launch (functionPtr, 0x4000, "PTV_Application", priority))
        priority++;
```

}

See Also

pk_LaunchNotify, pk_LaunchThread

pk_LaunchNotify

Creates a new thread. When the thread is deleted, a notification event is sent to the specified queue.

Syntax

```
Boolean pk_LaunchNotify(void (*code)(void *d),
   size_t stk_size, void *data, ui32 priority,
   queue *notify);
```

Parameters

→ *code*	A pointer to the code to be executed by the new thread.
→ *d*	A pointer to a parameter for *code*.
→ *stk_size*	The size, in bytes, of the stack to allocate for this thread.
→ *data*	A pointer to a private data area.
→ *priority*	The thread priority. The lower 5 bits represents the priority, which can range from 0 to 31.
→ *notify*	Optionally, a pointer to a queue to receive a kEt_pkThreadExit event on exit.

Returns

TRUE if the thread was launched successfully, and FALSE if the specified priority is already in use by another thread.

Comments

No function exists for deleting the thread because the thread is automatically deleted when the function specified by *code* returns.

Exceptions

kPtv_MemoryFullErr
kPtv_ParamErr

Example

None

See Also

pk_Launch, pk_LaunchThread

pk_LaunchThread

Creates a new thread and returns the thread identifier. When the thread is deleted, a notification event is sent to the specified queue.

Syntax

```
ui32 pk_LaunchThread(void (*code)(void *d),
   size_t stk_size, void *data, ui32 app_priority,
   queue *notify);
```

Parameters

→ *code*	A pointer to the code to be executed by the new thread.
→ *d*	A pointer to a parameter for *code*.
→ *stk_size*	The size, in bytes, of the stack to allocate for the new thread.
→ *data*	A pointer to a private data area.
→ *app_priority*	The application execution priority. The lower 3 bits represents the priority, which can range from 1 (highest) to 7 (lowest).
→ *notify*	An optional pointer specifying a queue to receive the kEt_PkThreadExit event when the thread has run to completion.

Returns

The thread identifier: a number from 1 to 31.

Comments

No function exists for deleting the thread because the thread is automatically deleted when the function specified by *code* returns.

Exceptions

kPtv_BusyErr
kPtv_MemoryFullErr
kPtv_ParamErr

Example

None

See Also

pk_Launch, pk_LaunchNotify

pk_Lock

Locks the kernel.

Syntax

```
void pk_Lock();
```

Parameters

None

Returns

None

Comments

This function disables interrupts, thereby preventing context switches.

Exceptions

None

Example

None

See Also

pk_Unlock

pk_MakeEvent

Creates an event, but does not deliver it.

Syntax

```
event * pk_MakeEvent(ui32 code, i32 x, i32 y, i32 z);
```

Parameters

→ *code*	The event type code.
→ *x*	An optional event data field.
→ *y*	An optional event data field.
→ *z*	An optional event data field.

Returns

A pointer to the new event.

Comments

None

Exceptions

kPtv_MemoryFullErr

Example

None

See Also

pk_DeliverEvent, pk_PostEvent, pk_ScheduleEvent

pk_NewQueue

Creates a new event queue.

Syntax

```
queue * pk_NewQueue();
```

Parameters

None

Returns

A pointer to the new queue, or NULL if there was insufficient memory from which to allocate a new queue.

Comments

None

Exceptions

kPtv_MemoryFullErr

Example

None

See Also

pk_DeleteQueue, pk_FlushQueue

pk_NextEvent

Gets the next event from the specified queue.

Syntax

```
event * pk_NextEvent(queue *q,
   const TimeValue timeout);
```

Parameters

→ *q*	The queue from which to retrieve the next event.
→ *timeout*	The time at which to return if an event has still not been delivered to the queue specified by *q*. An application can also specify kPtv_Forever to wait an indefinite period of time and return only when an event is delivered to the queue.

Returns

A pointer to the next event, or NULL if a timeout occurred (signalling that the queue is still empty).

Comments

None

Exceptions

None

Example

This example waits for an event.

```
testEvent = pk_NextEvent (myQueue, kPtv_Forever);
```

See Also

pk_NextEventCritical, pk_NextEventImmediate, pk_ReturnEvent

pk_NextEventCritical

Gets the next event from the specified queue, and upgrades the calling thread's condition to critical at some point.

Syntax

```
event * pk_NextEventCritical(queue *q,
   const TimeValue timeout, const TimeValue deadline);
```

Parameters

→ *q*	The queue from which to retrieve the next event.
→ *timeout*	The amount of time to wait for an event before returning.
→ *deadline*	When to upgrade the calling thread's condition to critical.

Returns

A pointer to the next event, or NULL if a timeout occurred (signalling that the queue is still empty).

Comments

None

Exceptions

None

Example

None

See Also

pk_NextEvent, pk_NextEventImmediate, pk_ReturnEvent

pk_NextEventImmediate

Gets the next event from the specified queue, returning immediately if the queue is empty.

Syntax

```
event * pk_NextEventImmediate(queue *q);
```

Parameters

→*q*	The queue from which to retrieve the next event.

Returns

A pointer to the next event, or *NULL* if the queue is empty.

Comments

None

Exceptions

None

Example

None

See Also

pk_NextEvent, pk_NextEventCritical, pk_ReturnEvent

pk_PeekEvent

Returns a pointer to the specified event in a queue.

Syntax

```
event * pk_PeekEvent(queue *q, ui32 n);
```

Parameters

→ *q*	A pointer to the queue in which the event resides.
→ *n*	The event number to which to return a pointer (1 for the first, 2 for the second, and so on).

Returns

A pointer to the specified event, or NULL if the event does not exist.

Comments

Handling a pointer to an event can be tricky if other threads are servicing the queue. We recommend that you lock threads as necessary.

Exceptions

None

Example

None

See Also

pk_CopyEvent

pk_PostEvent

Posts an event.

Syntax

```
void pk_PostEvent(ui32 code, i32 x, i32 y, i32 z);
```

Parameters

→ *code*	The event type code.
→ *x*	An optional event data field.
→ *y*	An optional event data field.
→ *z*	An optional event data field.

Returns

None.

Comments

The event type (in conjunction with the mask) is checked against each event interest in the order in which the event interests were registered. When a match is found, the event interest is triggered.

If a filter was specified, the filter procedure is called and the return code determines whether a copy of the event is sent to the queue. If no filter was specified, the event is delivered directly to the queue.

Posting an event may trigger multiple event interests and result in multiple copies of the event being delivered to multiple queues.

Exceptions

kPtv_MemoryFullErr

Example

```
void EventPoster (void *data)
{
    ui32    dCustomEvent = 0x29000000;

    do
    {
        // Every 5 seconds post an event
        // Alternate between 'posting' and 'delivering'
```

```
            pk_SleepFor (1000 * 25000); // Sleep for one second

            // Broadcast an event, including some arbitrary data
            pk_PostEvent (dtCustomEvent, 0x8, 0x29, 0x85);

            pk_SleepFor (1000 * 25000); // Sleep for one second

            // Post an event to a specified queue
            pk_DeliverEvent (myQueue, dtCustomEvent, 0x12, 0x14, 0x92);

        } while (1);
    }
```

See Also

pk_DeliverEvent, pk_ScheduleEvent

pk_RegisterInterest

Registers an interest in a particular class of events.

Syntax

```
ei * pk_RegisterInterest(ui32 code, ui32 mask,
   Boolean (*filter)(event *e), queue *q);
```

Parameters

→ *code*	The event type in which you are interested.
→ *mask*	An event mask which further clarifies events of interest.
→ *filter*	The interrupt service routine (ISR) to be called when the event is created. This procedure takes a pointer to the event as an argument, and returns a boolean value.
→ *queue*	A pointer to the queue that receives events.

Returns

A pointer to the event interest.

Comments

An event interest specifies a type and mask which determine the class of events to watch for.

You can supply a filter procedure that is called immediately when the event is posted and that operates in the context of the event creator (which may be an ISR).

An event interest may have only a filter or only a queue, or it may have both.

Exceptions

kPtv_MemoryFullErr

Example

```
// Register interest in several sources
pk_RegisterInterest (kDt_Controller | kDt_Player1, kEt_Any & kEd_Any, 0,
                     myQueue);
```

```
pk_RegisterInterest (kDt_Remote | kDi_Player1, kEt_Any & kEd_Any, 0,
                     myQueue);

pk_RegisterInterest (kDt_Console | kDi_Player1, kEt_Any & kEd_Any, 0,
                     myQueue);

pk_RegisterInterest (kDt_System | kEt_PkThreadExit, kEd_Any, 0,
                     myQueue);

pk_RegisterInterest (dtCustomEvent, kEt_Any & kEd_Any, 0, myQueue);
```

See Also

pk_RemoveInterest

pk_RemoveInterest

Removes an event interest.

Syntax

```
void pk_RemoveInterest(ei *interest);
```

Parameters

→ *interest* A pointer to an event interest.

Returns

None

Comments

If the value of *interest* is not valid, it is assumed to have been already removed.

Exceptions

None

Example

None

See Also

pk_RegisterInterest

pk_Rethrow

Passes the exception raised by pk_Throw from inside a catch block to an outer try block for continued processing.

Syntax

```
pk_Rethrow ();
```

Parameters

None

Comments

This is a macro. pk_Rethrow is equivalent to calling pk_Throw (pk_CurrentException).

This macro can be thought of as an alternative return mechanism. It passes control to the outer try block so that the exception can be processed. Control does not return to the location where the exception occurred.

Example

None

See Also

pk_Throw, pk_ThrowIfNull

pk_ReturnEvent

Returns an event to a queue.

Syntax

```
void pk_ReturnEvent(queue *q, event *e);
```

Parameters

→ *q*	A pointer to an event queue.
→ *e*	A pointer to the event to return.

Returns

None

Comments

Use pk_ReturnEvent primarily for implementing semaphores. Certain events, such as semaphore events, have limited interest and can be routed to, at most, one queue. When retrieving such events for processing, we recommend using pk_ReturnEvent, which routes the actual event rather than a copy of the event.

Exceptions

None

Example

None

See Also

pk_DeliverEvent, pk_NextEvent, pk_NextEventCritical, pk_NextEventImmediate

pk_ReturnInTry

Executes a C **return** call from within a try block after popping the try block off the exception handling stack.

Syntax

```
pk_ReturnInTry(ret);
```

Parameters

→ *ret* The return value.

Comments

This is a macro.

Attempts to use a standard return call from within a try block will result in an invalid exception stack, and any subsequent exception will cause control to be passed incorrectly to the catch blocks associated with the try block.

Example

None

See Also

pk_Try

pk_ScheduleDelivery

Arranges for the future delivery of an event.

Syntax

```
void pk_ScheduleDelivery(queue *where, TimeValue t,
   ui32 code, i32 x, i32 y, i32 z);
```

Parameters

→ *where*	The queue to which to deliver the event.
→ *t*	The delivery time.
→ *code*	The event type code.
→ *x*	An optional event data field.
→ *y*	An optional event data field.
→ *z*	An optional event data field.

Returns

None

Comments

None

Exceptions

kPtv_MemoryFullErr

Example

None

See Also

pk_DeliverEvent, pk_PostEvent, pk_ScheduleEvent

pk_ScheduleEvent

Posts an event at a future time.

Syntax

```
void pk_ScheduleEvent(TimeValue t, ui32 code, i32 x,
   i32 y, i32 z);
```

Parameters

→ *t*	The delivery time.
→ *code*	The event type code.
→ *x*	An optional event data field.
→ *y*	An optional event data field.
→ *z*	An optional event data field.

Returns

None

Comments

If the time has already passed when pk_ScheduleEvent is called, the event is delivered immediately.

Exceptions

kPtv_MemoryFullErr

Example

None

See Also

pk_DeliverEvent, pk_PostEvent, pk_ScheduleDelivery

pk_SleepFor

Sleeps (blocks) for a specified number of clock ticks.

Syntax

```
void pk_SleepFor(ui32 clockticks);
```

Parameters

→ *clockticks* The number of clock ticks to block for.

Returns

None

Comments

This function implements a timeout with an accuracy of +/- 5 milliseconds.

Exceptions

None

Example

```
void EventPoster (void *data)
{
    ui32    dtCustomEvent = 0x29000000;

    do
    {
        // Every 5 seconds post an event
        // Alternate between 'posting' and 'delivering'

        pk_SleepFor (1000 * 25000);  // Sleep for one second

        // Broadcast an event, including some arbitrary data
        pk_PostEvent (dtCustomEvent, 0x8, 0x29, 0x85);

        pk_SleepFor (1000 * 25000);  // Sleep for one second
```

```
            // Post an event to a specified queue
            pk_DeliverEvent (myQueue, dtCustomEvent, 0x12, 0x14, 0x92);

        } while (1);
    }
```

See Also

pk_CpuTime

pk_StackAvail

Determines how much of a thread's allocated stack is left.

Syntax

```
pk_StackAvail(threadID);
```

Parameters

→ *threadID*	A thread identifier, returned by one of the pk_Launch* functions.

Comments

This is a macro.

Example

None

See Also

pk_Launch, pk_LaunchNotify, pk_LaunchThread, pk_MyStackAvail, pk_StackSize

pk_StackProbe

Raises an exception if the stack allocated to a thread has been exceeded.

Syntax

```
pk_StackProbe(threadID);
```

Parameters

→ *threadID*	A thread identifier, returned by one of the pk_Launch* functions.

Comments

This is a macro.

Use this function during development to check whether or not a thread's allocated stack has been exceeded.

Exception

kPtv_StackOverflow

Example

None

See Also

pk_Launch, pk_LaunchNotify, pk_LaunchThread, pk_StackAvail, pk_StackSize

pk_StackSize

Determines how much stack was allocated for a thread.

Syntax

```
pk_StackSize(threadID);
```

Parameters

→ *threadID* A thread identifier, returned by one of the pk_Launch* functions.

Comments

This is a macro.

Example

None

See Also

pk_Launch, pk_LaunchNotify, pk_LaunchThread, pk_MyStackSize, pk_StackAvail

pk_TakeEvent

Takes an event from a queue.

Syntax

```
event * pk_TakeEvent(queue *q, ui32 n);
```

Parameters

→ *q*	A pointer to the queue from which to take the event.
→ *n*	The event number to take (1 for the first, 2 for the second, and so on).

Returns

A pointer to the event, or NULL if the event does not exist.

Comments

None

Exceptions

None

Example

None

See Also

pk_CopyEvent, pk_PeekEvent

pk_Throw

Raises an exception by passing the specified error to the system's exception handler.

Syntax

```
pk_Throw(type);
```

Parameters

→ *type* The exception type.

Comments

This is a macro.

This macro can be thought of as an alternative return mechanism. It passes control to the outer try block so that the exception can be processed. Control does not return to the location where the exception occurred.

Example

None

See Also

pk_Rethrow, pk_ThrowIfNull

pk_ThrowIfNull

Raises a kPtv_MemoryFullErr exception to the system's exception handler, if the pointer is null.

Syntax

```
pk_ThrowIfNull (void *myPointer);
```

Parameters

→ *myPointer* A null pointer.

Comments

This is a macro.

This macro can be thought of as an alternative return mechanism. If *myPointer* is NULL, it passes control to the outer try block so that the exception can be processed. Control does not return to the location where the exception occurred.

pk_ThrowIfNull is useful after any call that creates

Example

```
pk_Try
{
    // Create a new screen
    gScreenID = scr_NewSharedScreen (appID,
                    kScr_ScreenMode_LowResolutionNTSC);
    pk_ThrowIfNull (gScreenID);
    .
    .
}
pk_Catch (kPk_ExceptionAll)
{
    // Exception processing code
}
pk_EndCatch
```

See Also

pk_Rethrow, pk_Throw

pk_Try

Delimits a block of code in which to catch exceptions.

Syntax

```
pk_Try
```

Comments

This is a macro.

This function delimits a block of code known as a *try block*. One or more catch blocks must follow the try block and define the actual processing functionality for the exception.

The scope of the try block determines its priority. During exception processing, inner try blocks take precedence over outer ones.

A try block is pushed onto the exception stack. All exceptions are thrown to this try block's catch routines or to that of later try blocks until execution of this try block completes.

Example

This is a very simple example of an exception handler. The try block surrounds the PlayHighChimeSound routine, and a single catch block catches all exceptions.

```
void PlayHighChimeSound (void)
{
    ul32 playerPtr;

    pk_Try
    {
        // Create an audio player that deletes the resources upon
        // completion of playing the sound.
        playerPtr = ap_NewRAMAIFF ((ul8 *)High_Chime,
                    High_Chime_size, TRUE);
        // Start playing the sound
        ap_play (playerPtr);
    }
    // Catch all audio exceptions here.
    pk_Catch (kPk_ExceptionAll)
    {
```

```
            printf ("Audio error caught in PlayHighChimeSound\n");
        }
        pk_EndCatch
    }
```

See Also

pk_ReturninTry

pk_Unlock

Unlocks the kernel.

Syntax

```
void pk_Unlock();
```

Parameters

None

Returns

None

Comments

This function re-enables interrupts, thereby enabling context switches.

Exceptions

None

Example

None

See Also

pk_Lock

CLAIMS

1. An operating system kernel for execution on a home communication terminal (HCT) on which a plurality of different events can occur and a plurality of different threads can be executed, the operating system kernel comprising:

an event registering function which allows each of the plurality of different threads to register interest in one of the plurality of different events by specifying parameters which indicate qualifications for the one event of interest;

a data structure comprising a list of event objects each comprising one or more parameters specified using the event registering function; and

an event handler which, responsive to receiving an event descriptor corresponding to one of the plurality of different events, traverses the data structure, compares the event descriptor to one or more of the parameters in each of the event objects and, responsive to a determination that the descriptor matches the one or more parameters in a particular one of the event objects, provides at least part of the event descriptor to a thread having an interest registered in the one event.

2. The kernel according to claim 1, wherein each event object comprises at least two parameters, a first parameter comprising a code field and a second parameter comprising a mask field, wherein the mask field indicates which of a plurality of fields in the event descriptor are germane, and wherein the code field indicates a value that each germane field in the event descriptor must match.

3. The kernel according to claim 2, wherein the event handler multiplies the mask field with at least part of the event descriptor and compares the result with the code field in order to perform the comparison.

4. The kernel according to claim 3, wherein the event handler places the at least portion of the event descriptor on a queue designated in the event object.

5. The kernel according to claim 1, wherein the one or more parameters indicates a device type and an event type to be qualified.

6. The kernel according to claim 1, wherein each of the one or more parameters indicates a data value representing a key press from a device coupled to the HCT.

7. The kernel according to claim 1, wherein the one or more parameters comprises a filter to be executed by the event handler to further qualify the event descriptor.

8. The kernel according to claim 7, wherein the filter is executed in the context of the kernel rather than in the context of a thread.

9. The kernel according to claim 7, wherein the filter checks a position of a cursor with reference to a window in order to further qualify the event.

10. The kernel according to claim 7, wherein the event handler compares the event descriptor with the one or more parameters in each of the event objects and, responsive to a determination that the descriptor matches the one or more parameters in a particular one of the event objects, executes the filter prior to providing the at least part of the event descriptor to the thread having an interest

registered in the one event.

11. The kernel according to claim 1, further comprising an event deregistration function which allows each of the plurality of different threads to remove a previously registered event interest.

12. A method of filtering events in an operating system kernel executing on a home communication terminal (HCT) having a plurality of threads, comprising the steps of:

(1) registering interest in one or more events from one of the plurality of threads by creating a data structure comprising one or more parameters which limit the type of events which are to be provided to the one thread;

(2) detecting that an event has occurred in the system and, responsive thereto, creating an event descriptor which describes details of the event;

(3) in the operating system kernel, comparing the event descriptor created in step (2) with the one or more parameters in the data structure created in step (1) and, responsive to a determination that the event descriptor matches the event, providing at least part of the event descriptor to a thread corresponding to the data structure.

13. The method of claim 12, wherein step (1) comprises the step of creating a data structure comprising a code field and a mask field, wherein the mask field indicates which portions of the event descriptor are germane to the comparison in step (3), and wherein the code field indicates a value that each germane portion in the event descriptor must match.

14. The method of claim 12, wherein step (1) comprises the step of using parameters which indicate a device type and an event type to be qualified.

15. The method of claim 12, wherein step (1) comprises the step of specifying a filter to be executed by the kernel to further qualify the event descriptor, and wherein step (3) comprises the step of executing the specified filter prior to providing at least part of the event descriptor to the thread.

16. The method of claim 15, wherein step (3) comprises the step of executing a filter which modifies the at least part of the event descriptor prior to providing it to the thread.

17. A method of implementing on a home communication terminal (HCT) a semaphore function using an operating system kernel which lacks semaphore functions, comprising the steps of:

(1) creating a queue for a shared resource using a kernel-supplied queue creation function which is accessible to threads executing on the HCT;

(2) storing an event object on the queue, using a kernel-supplied object delivery function which is accessible to threads executing on the HCT, to indicate the availability of the shared resource;

(3) from a first thread, executing a kernel-supplied object retrieval function which retrieves the event object from the queue, wherein if the event object is not on the queue the first thread is suspended by the kernel;

(4) from a second thread, executing the kernel-supplied object retrieval function to attempt retrieval of the event object from the queue, and, upon failure

to retrieve the event object from the queue, suspending thereby the second thread; and

(5) from the first thread, executing a kernel-supplied object release function which returns the event object back to the queue, and thereafter resuming operation of the object retrieval function started in step (4).

18. The method of claim 17, wherein step (2) comprises the step of storing a plurality of event objects on the queue, each corresponding to one of a plurality of shared resources.

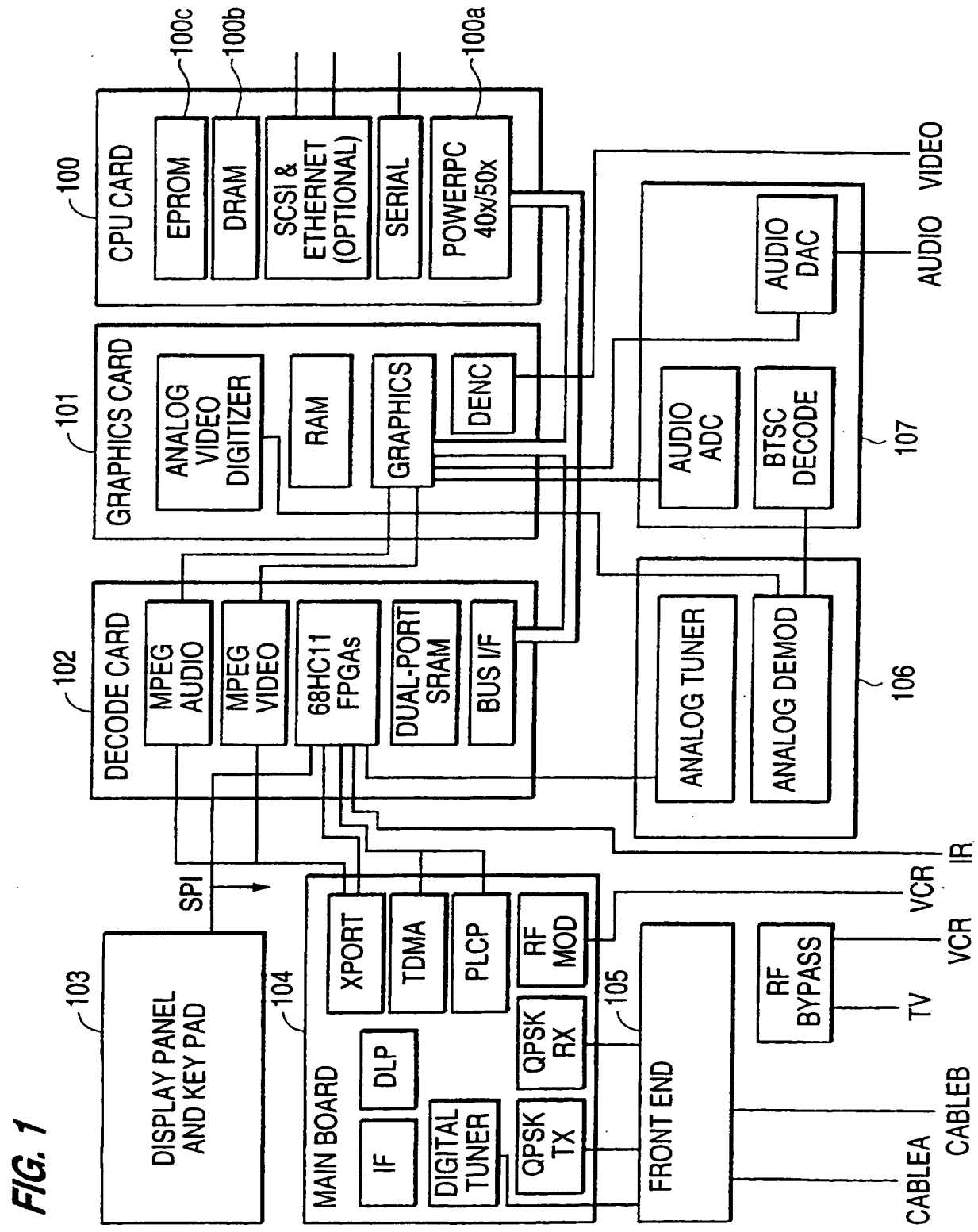


FIG. 1

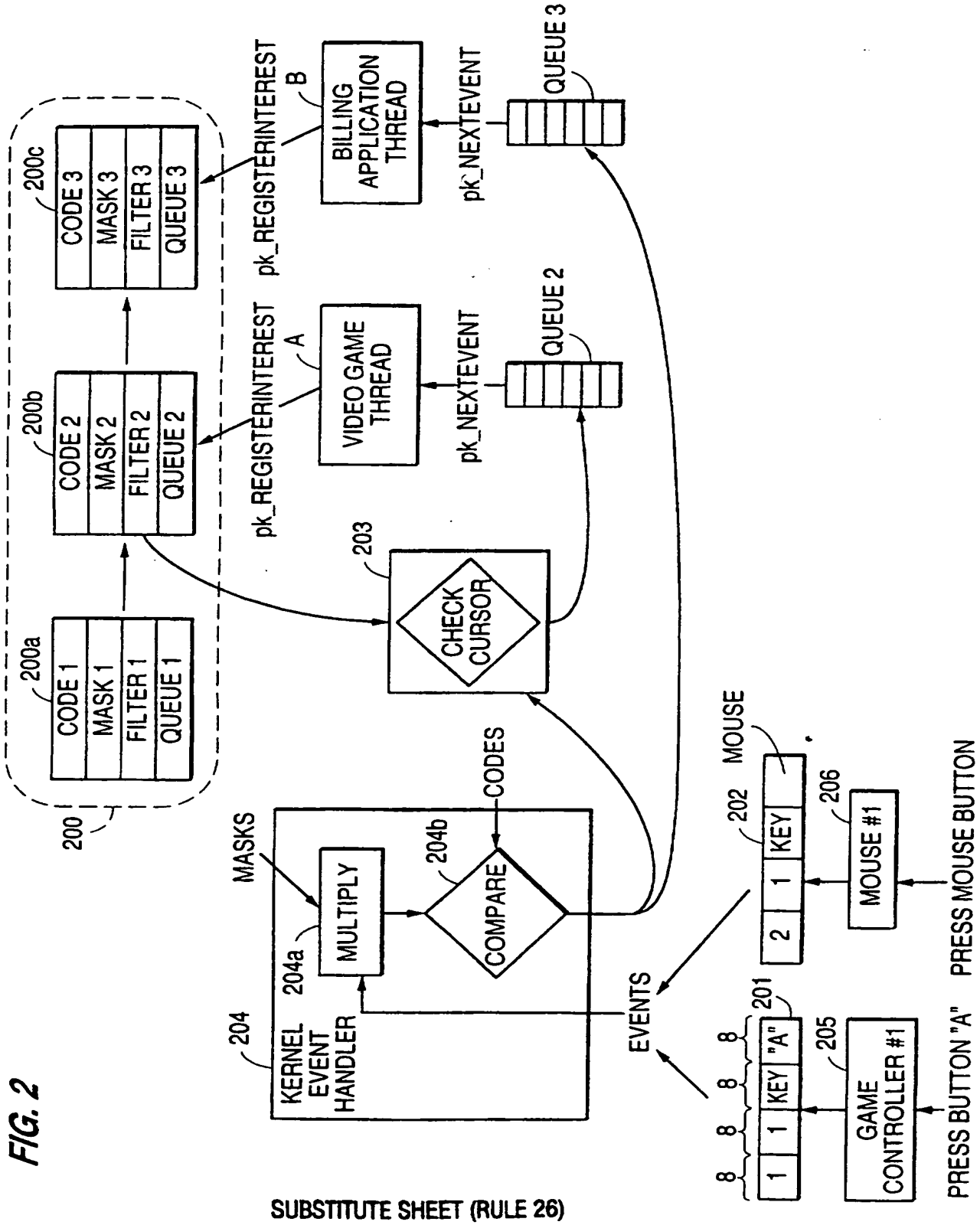


FIG. 2

FIG. 3

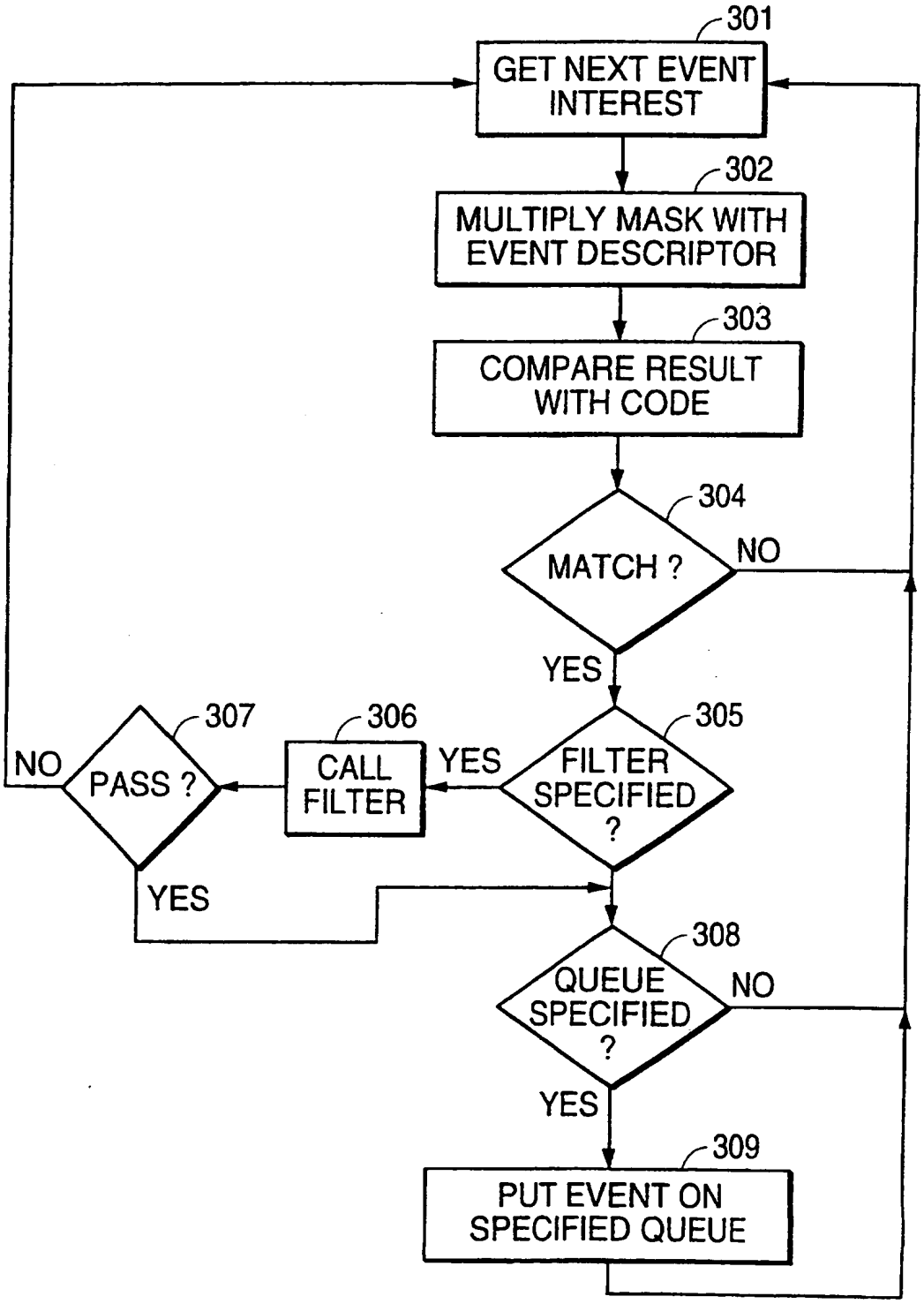


FIG. 4

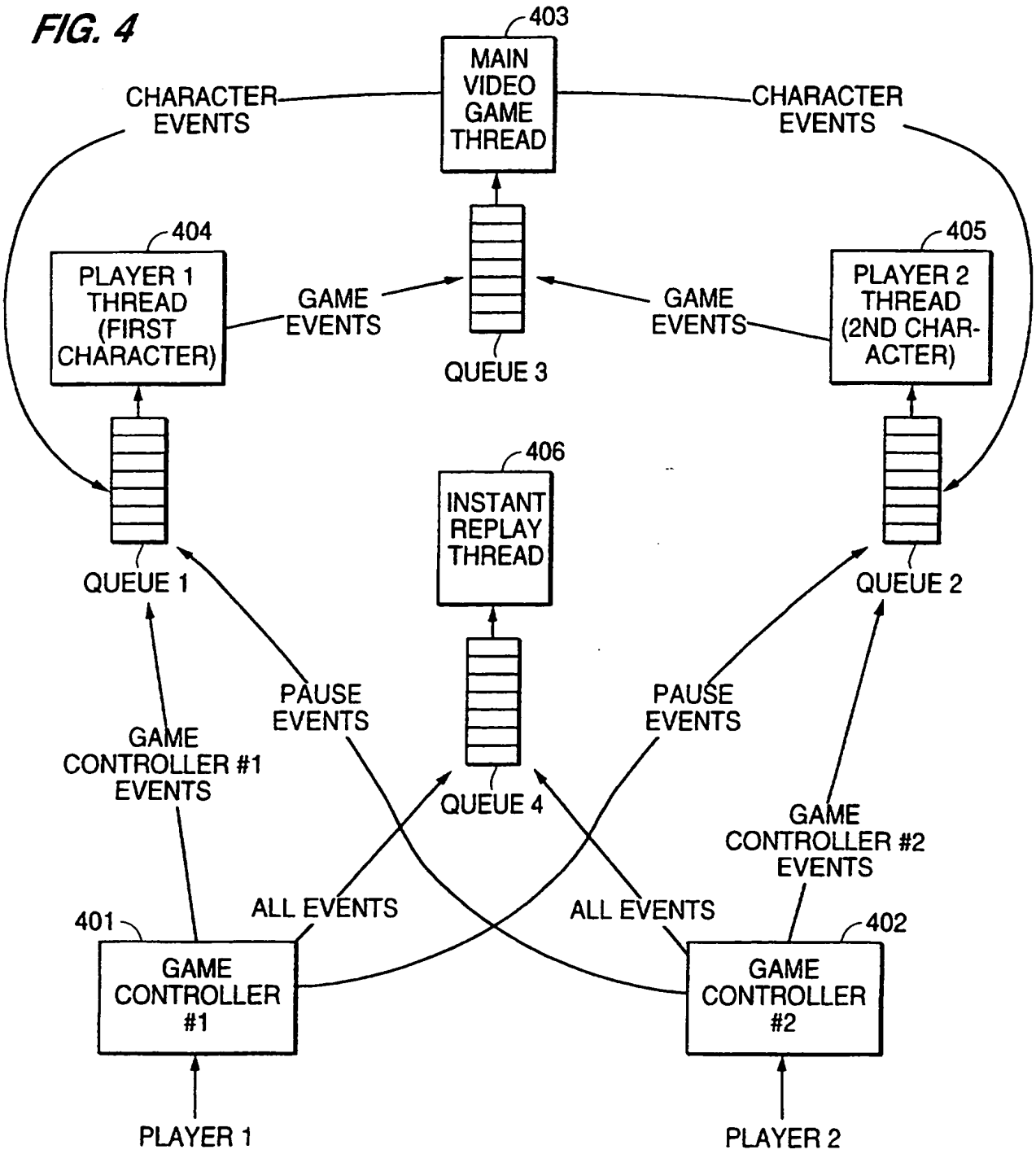


FIG. 5

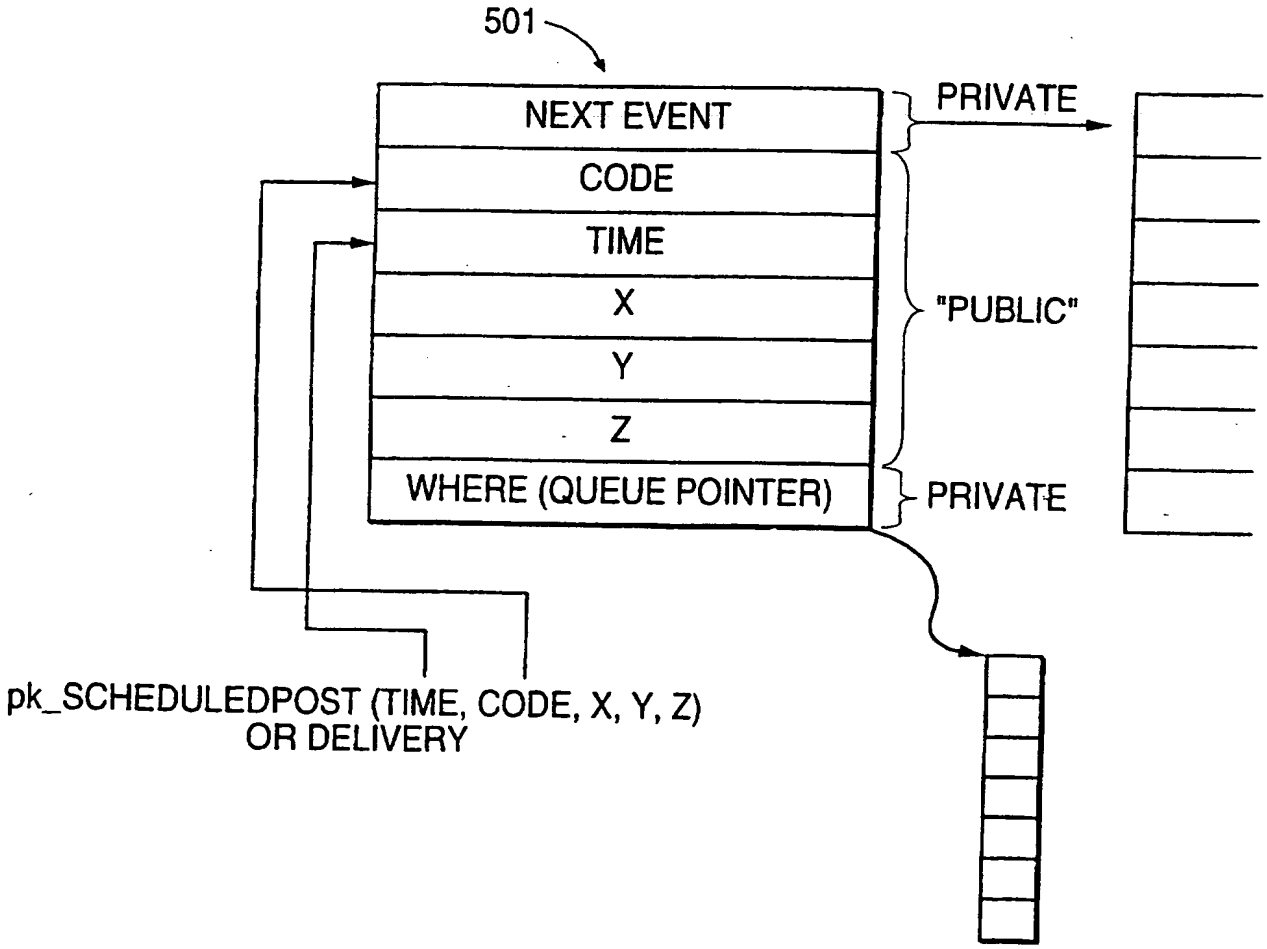
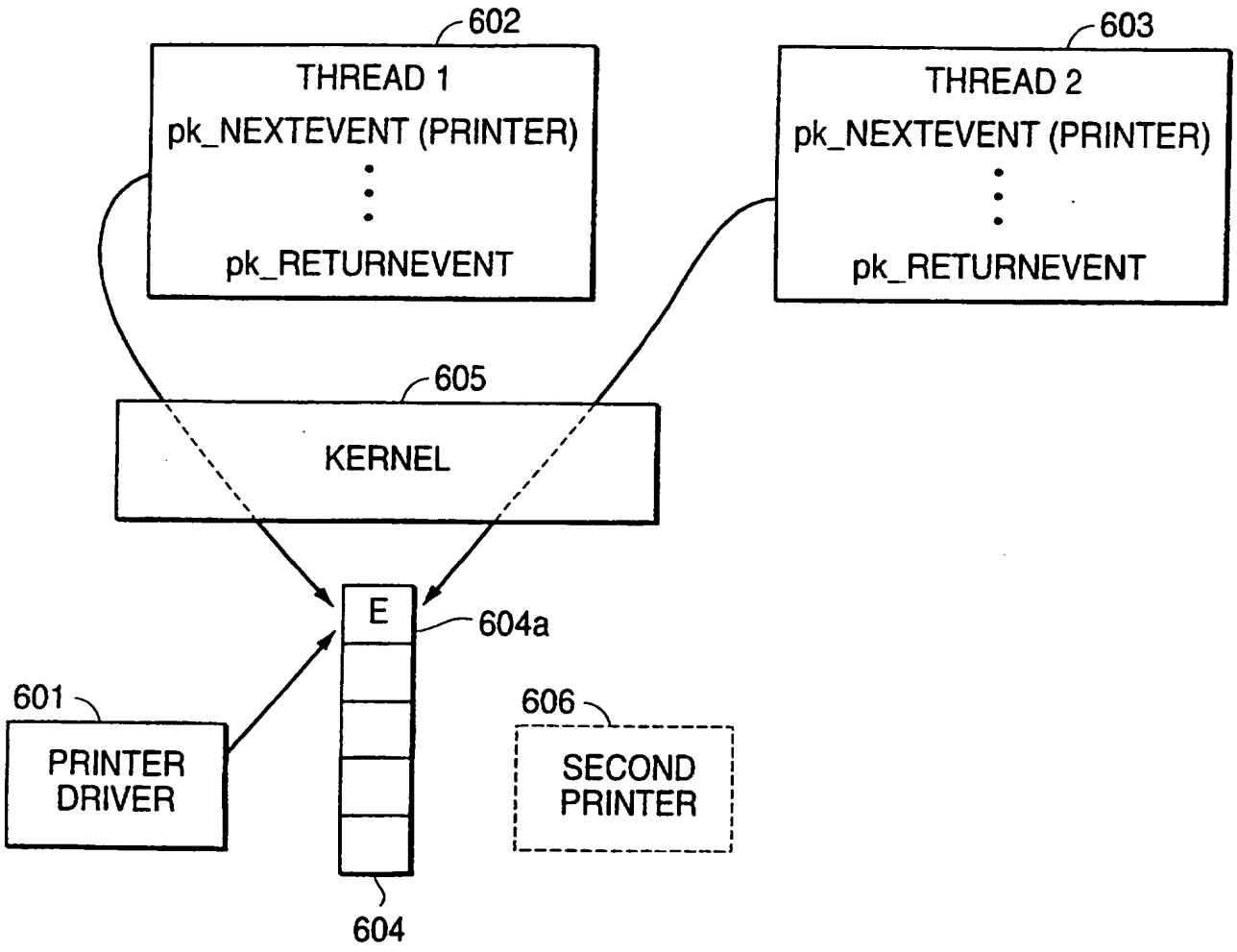


FIG. 6



INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/20126

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : G06F 11/30, 9/00
US CL : 395/670, 677

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/670, 677

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
	Please See Continuation of Second Sheet.	

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 08 APRIL 1997	Date of mailing of the international search report 12 JUN 1997
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer Sue Lao Telephone No. (703) 305-9657

Form PCT/ISA/210 (second sheet)(July 1992)*

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/20126

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,430,875 A (MA) 04 July 1995, see entire document	1-16
Y	US 5,321,837 A (DANIEL et al) 14 June 1994, figure 12, column 5, lines 56-65	2, 4, 13
Y	US 5,465,335 A (ANDERSON) 07 November 1995, column 2, lines 16-20, column 6, lines 52-59	3, 16
Y	US 5,301,270 A (STEINBERG et al) 05 April 1994, column 26, lines 46-49, column 27, lines 9-20, figure 30	11
Y	US 5,325,536 A (CHANG et al) 28 June 1994, column 3, lines 50-54	8
A, P	US 5,566,337 A (SZYMANSKI et al) 15 October 1996, entire document	1-16
Y	US 5,428,781 A (DUAULT et al) 27 June 1995, see entire document, especially, column 3, lines 1-31, column 4, lines 5-13	17-18